Scientific Data Storage

Valentin Haenel

Freelance Consultant and Software Developer
http://haenel.co

Feb 2013

Version: 2013-02-pcp13.1 https://github.com/pcp13/data-storage-talk

Outline

Outline

What does *serialization* mean?

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be resurrected later in the same or another computer environment.

The basic mechanisms are to flatten object(s) into a one-dimensional stream of bits, and to turn that stream of bits back into the original object(s).

From: http://www.parashift.com/c++-faq-lite/serialization.html

Serialization tools

There are literally zillions of serialization tools and formats (text, *XML*, or binary based), but well be focusing on those that are:

- Easy to use
- Space-efficient
- Fast

In particular, we are not going to discuss text-based formats (e.g. *XML*, *CSV*, *YAML*, *JSON* ...).

Serialization tools that comes with Python

Python comes with a complete tool set of modules for serialization purposes:

- *pickle*, and its cousin, *cPickle*, for quick-and-dirty serialization
- *shelve*, a persistent dictionary based on DBM databases
- A common database API for communicating with relational databases

Serialization tools for binary data

Additionally, there are lots of third-party libraries for specialized uses. Here will center on numerical formats:

- *NPY*, *NPZ*: NumPy's own format
- Wrappers for *HDF5*, a standard de-facto format and library: *PyTables*, *h5py*

Outline

The pickle module

- Serializes an object into a stream of bytes
- can be saved to a file (or a string) and later restored

```
import pickle

filename, colors = '/tmp/ex_pickle', ['red', 'black']

with open(filename, 'wb') as f:
    pickle.dump(colors, f)

# ... later on
with open(filename, 'rb') as f:
    obj = pickle.load(f)

assert colors == obj
```

What does `pickle` do

- It can serialize both basic Python data structures or user-defined classes.
- Always serializes data, not code (it tries to import classes if found in the pickle).

Warning

For security reasons, programs should **NEVER** unpickle data received from untrusted sources.
See also: http://nedbatchelder.com/blog/201302/war_is_peace.html

Its cPickle cousin

- Implemented in C (i.e. significantly faster than `pickle`).
- But more restrictive (does not allow subclassing of the `Pickler` and `Unpickler` objects).
- Python 3 `pickle` can use the C implementation transparently.

Picklin' a Numpy Array

```
>>> a = np.linspace(0, 100, 1e7)
>>> %timeit pickle.dump(a, open('p1','w'))
1 loops, best of 3: 8.92 s per loop
>>> %timeit pickle.dump(a, open('p2','w'), pickle.HIGHEST_PROTOCOL)
1 loops, best of 3: 509 ms per loop
>>> ls -sh p1 p2
186M p1   77M p2
```

Always try to use `cPickle` and `HIGHEST_PROTOCOL`

`pickle/cPickle` limitations and recommendations

- You need to reload all the data in the pickle before you can use any part of it. That might be inconvenient for large datasets.
- Data can only be retrieved by other Python interpreters. You loose data portability with other languages.
- Not every object in Python can be serialized by `pickle` (e.g. extensions).

Recommendations for using `pickle`

- Use it mainly for small data structures.
- If you have a lot of variables that you want to save, use a dictionary for tying them together first.
- When using Ipython, be sure to use the very convenient `%store` magic (it uses `pickle` under the hood).

Outline

The `shelve` module

- Provides support for persistent objects using a special *shelf* object.
- The *shelf* behaves like a disk-based dictionary or *key-value store*.
- The values of the dictionary can be any object that can be pickled.

Example

```
import shelve

db = shelve.open("database", "c")
db["one"] = 1
db["two"] = 2
db["three"] = 3
db.close()

db = shelve.open("database", "r")
for key in db.keys():
    print repr(key), repr(db[key])
```

```
$ python code/ex_shelve.py
'one' 1
'three' 3
'two' 2
```

Pros and cons of the `shelve` module

Pros

- Easy to retrieve just a selected set of variables.
- Specially handy for large pickles.

Cons

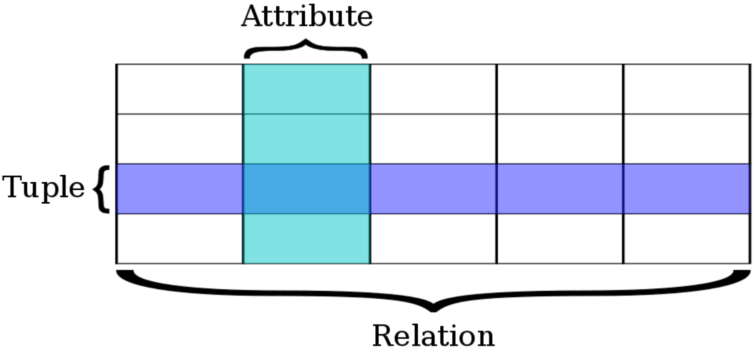
- Suffers the same problems as `pickle`.

Outline

What's a relational database?

- A set of tables containing data fitted into predefined categories.
- Each table (a relation) contains one or more data categories in columns.
- Each row contains a unique instance of data for the categories defined by the columns.
- Data can be accessed in many different ways without having to reorganize the tables.

Terminology



Base and derived relations

- In a relational database, all data are stored and accessed via relations.
- Relations that store data are called *base relations*, and in implementations are called *tables*
- Other relations do not store data, but are computed by applying relational operations to other relations.
- These relations are sometimes called *derived relations*
- In implementations these are called *views* or *queries*

Example of relational database

PubID	Publisher	PubAddress
03-4472822	Random House	123 4th Street, New York
04-7733903	Wiley and Sons	45 Lincoln Blvd, Chicago
03-4859223	O'Reilly Press	77 Boston Ave, Cambridge
03-3920886	City Lights Books	99 Market, San Francisco

AuthorID	AuthorName	AuthorBDay
345-28-2938	Haile Selassie	14-Aug-92
392-48-9965	Joe Blow	14-Mar-15
454-22-4012	Sally Hemmings	12-Sept-70
663-59-1254	Hannah Arendt	12-Mar-06

ISBN	AuthorID	PubID	Date	Title
1-34532-482-1	345-28-2938	03-4472822	1990	Cold Fusion for Dummies
1-38482-995-1	392-48-9965	04-7733903	1985	Macrame and Straw Tying
2-35921-499-4	454-22-4012	03-4859223	1952	Fluid Dynamics of Aquaducts
1-38278-293-4	663-59-1254	03-3920886	1967	Beads, Baskets & Revolution

Queries with SQL language

Simple query involving one single table (relation):

```
SELECT AuthorName FROM AUTHORS WHERE AuthorBDay > 1970
```

Complex query involving multiple relations:

```
SELECT AuthorName FROM AUTHORS a, BOOKS b, PUBLISHERS p
    WHERE AuthorBDay > 1970
        AND a.AuthorID = b.AuthorID
        AND b.PubID = p.PubID
        AND p.Publisher = "Random House"
    GROUP BY AuthorBDay
```

Beware

complex queries can consume a lot of resources!

Relational database API specification

- The Python community has developed a standard API for accessing relational databases in a uniform way (PEP 249).
- Specific database modules (e.g. MySQL, Oracle, Postgres ...) follow this specification, but may add more features.
- Python comes with SQLite.
  - SQL Database, but stored as a single file on disk.
  - A relational database accessible via the `sqlite3` module.

ORM (Object Relational Mapping)

- The relational database API in Python is powerful, but pretty rough to use and **not** object-oriented.
- Many projects have appeared to add an object-oriented layer on top of this API:
  - SQLAlchemy
  - Django's native ORM
  - Storm
  - Elixir
  - SQLObject (the one that started it all)
  - ... probably a lot more ...

Define Objects

```
from storm.database import  create_database
from storm.store import  Store
from storm.locals import Int, Unicode, Reference

class Kind(object):
    __storm_table__ = 'kinds'
    id = Int(primary=True)
    name = Unicode()

class Thing(object):
    __storm_table__ = 'things'
    id = Int(primary=True)
    name = Unicode()
    kind_id = Int()
    kind = Reference(kind_id, Kind.id)
```

Setup database

```
db = create_database('sqlite:')
store = Store(db)

store.execute("CREATE TABLE kinds  "
                      "(id INTEGER PRIMARY KEY, name VARCHAR)")
store.execute("CREATE TABLE things  "
                      "(id INTEGER PRIMARY KEY, name VARCHAR, kind_id IN
```

Add Flowers

```
flowers = Kind()
flowers.name = u"Flowers"
store.add(flowers)

red_rose = Thing()
red_rose.name = u'Red Rose'
red_rose.kind = flowers
store.add(red_rose)

violet = Thing()
violet.name = u'Violet'
violet.kind = flowers
store.add(violet)
```

Add Vases and commit

```
vases = Kind()
vases.name = u"Vases"
store.add(vases)

amphora = Thing()
amphora.name= u'Amphora'
amphora.kind = vases;
store.add(amphora)

store.commit()
```

Search and Retrieve

```
all_flowers = store.find((Kind, Thing),
        Thing.kind_id == Kind.id,
        Kind.name == u'Flowers')

print [(kind.name, thing.name) for kind, thing in all_flowers]

all_vases = store.find((Kind, Thing),
        Thing.kind_id == Kind.id,
        Kind.name == u'Vases')

print [(kind.name, thing.name) for kind, thing in all_vases]
```

Executing

```
$ python code/ex_storm.py
[(u'Flowers', u'Red Rose'), (u'Flowers', u'Violet')]
[(u'Vases', u'Amphora')]
```

RDBMs highlights

They offer *ACID* (atomicity, consistency, isolation, durability) properties, that can be translated into:

- Referential integrity.
- Transaction support.
- Data consistency.

$+$ Indexing capabilities (accelerate queries in large tables).

But this comes with a price...

RDBMs drawbacks

- Insertions are SLOOOW.
- Not very space-efficient.
- Not well adapted to handle large numerical datasets (no direct interface with NumPy).
- You need a knowledgeable RDBM administrator to squeeze all the performance out of them.

Outline

What's a numerical binary format?

- It is a format specialized in saving and retrieving large amounts of numerical data.
- Usually come with libraries that can understand that format.
- They range from the very simple (NPY) to rather complex and powerful (HDF5).
- There are a really huge number of numerical formats depending on the needs. Will focus on just on few.

Why we need a binary format?

- They are closer to memory representation.
- Their representation is space-efficient (1 byte in-memory $\approx$ 1 bytes on disk).
- They are CPU-friendly (in general you do not have to convert from one representation to another).

NumPy: the real cornerstone of numerical interfaces

- NumPy is the standard de-facto for dealing with numerical data in-memory.
- Hence, most of the interfaces to numerical formats in the Python world use NumPy to interact with the database.
- In some cases the integration is so tight that it could be difficult to say if you are working with NumPy or the interface.

The NPY format

- Created back in 2007 for overcoming limitations of `pickle` for NumPy arrays as well as `numpy.tofile()` / `numpy.fromfile()` functions
- It is a binary format, so it is space-efficient.
- It comes integrated with NumPy.
- See also: A Simple File Format for NumPy Arrays

NPY exposes the simplest API for NumPy

```
import numpy as np

data = np.arange(1e7)
np.save('test.npy', data)
data2 = np.load('test.npy')

assert np.alltrue(data == data2)
```

What is in the file?

```
$ head -c 100 test.npy
NUMPYF{'descr': '<f8', 'fortran_order': False, 'shape': (10000000,), }
\T1\dh ?%
$ head -c 100 test.npy | xxd
0000000: 934e 554d 5059 0100 4600 7b27 6465 7363  .NUMPY..F.{'desc
0000010: 7227 3a20 273c 6638 272c 2027 666f 7274  r': '<f8', 'fort
0000020: 7261 6e5f 6f72 6465 7227 3a20 4661 6c73  ran_order': Fals
0000030: 652c 2027 7368 6170 6527 3a20 2831 3030  e, 'shape': (100
0000040: 3030 3030 302c 292c 207d 2020 2020 200a  00000,), }     .
0000050: 0000 0000 0000 0000 0000 0000 0000 f03f  ...............?
0000060: 0000 0000                                ....
```

... just a header plus binary ...

Memory-mapping and NPY

You can open a NPY file in memmap-mode for accessing data directly from disk:

```
>>> mmdata = np.load('test.npy', mmap_mode='r+')
>>> mmdata
memmap([  0.00000000e+00,   1.00000000e+00,   2.00000000e+00, ...,
         9.99999700e+06,   9.99999800e+06,   9.99999900e+06])
>>> mmdata[-10:] + mmdata[:10]
memmap([  9999990.,   9999992.,   9999994.,   9999996.,   9999998.,
         10000000.,  10000002.,  10000004.,  10000006.,  10000008.])
>>> del mmdata # close access to 'test.npy'
```

Saving several arrays with NPZ

The NPY format has a special mode that can save several arrays in one single ZIP file (but no compression is used at all!):

```
import numpy as np

a = np.linspace(0, 100, 1e7)
sina = np.sin(a)
np.savez('test.npz', a=a, sina=sina)
```

Just a ZIP file.

```
$ time python ex_npz.py
python ex_npz.py  1.50s user 2.18s system 98% cpu 3.754 total
$ file test.npz
test.npz: Zip archive data, at least v2.0 to extract
$ ls -sh test.npz
153M test.npz
```

Adding compression

```
$ time python ex_npzc.py
python ex_npzc.py  22.82s user 2.09s system 93% cpu 26.604 total
$ file testc.npz
testx.npz: Zip archive data, at least v2.0 to extract
$ ls -sh testc.npz
109M testc.npz
```

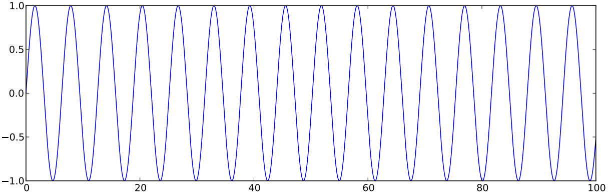
- The file is somewhat smaller
- ... but it takes much longer to save

- Uses the `DEFLATE` algorithm, which is optimizaed for text, not data

Loading several arrays with NPZ

```
>>> arrs = np.load('test.npz')
>>> arrs.items()
[('a',
  array([  0.00000000e+00,   1.00000010e-05,   2.00000020e-05, ...,
           9.99999800e+01,   9.99999900e+01,   1.00000000e+02])),
 ('sina',
  array([  0.00000000e+00,   1.00000010e-05,   2.00000020e-05, ...,
          -5.06382887e-01,  -5.06374264e-01,  -5.06365641e-01]))]

>>> pylab.plot(arrs['a'], arrs['sina'])
```



Pros and cons of NPY

Pros:

- Binary format, so space-efficient.
- Avoids duplication of data in memory during saving/loading operations.
- Array data accessible through memory-mapping.

Cons:

- The memory mapping feature only allows to deal with files that do not exceed the available virtual memory.
- Non-standard format outside the NumPy community.
- No other features than basic input/output (e.g. no metadata allowed).
- Has compression, but perhaps the wrong kind

Outline

The HDF5 format

- HDF5 (Hierarchical Data Format v5) is a library and file format for storing and managing any kind of data.
- http://www.hdfgroup.org/HDF5/doc/H5.format.html
- It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data.
- Originally developed at the NCSA, and currently maintained by The THG Group, a not for-profit organization.
- HDF5 has been around for over twenty years, and has become a standard de-facto format supported by many applications (MatLab, IDL, R, Mathematica ...).

Outstanding features of HDF5

- Can store all kinds of data in a variety of ways.
- Runs on most systems.
- Lots of tools to access data.
- Long term format support (HDF-EOS, CGNS).
- Library and format emphasis on I/O efficiency and different kinds of storage.

Python interfaces

- **h5py** is an attempt to map the HDF5 feature set to Python as closely as possible.
  - It also provides access to nearly all of the HDF5 C API (the so-called low-level API).
  - Not designed to go beyond HDF5/NumPy capabilities.
- **PyTables** builds up an additional abstraction layer on top of HDF5 and NumPy where it implements things like:
  - An enhanced type system (enumerated, time, variable length types and default values supported).
  - An engine for enabling complex queries and out-of-core computations (using Numexpr behind the scenes).
  - Advanced indexing capabilities (Optimally Partially Sorted Indices, OPSI)

Creating an HDF5 file

```
import tables
import numpy as np

f = tables.openFile("example.h5", "w")
group = f.createGroup("/", "reduced_data")
ds = f.createArray(group, "array", np.array([1, 2, 3, 4]))
```

```
>>> ds
/reduced_data/array (Array(4,)) ''
  atom := Int64Atom(shape=(), dflt=0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := None
```

Creating an Table

```
gen = ((i, i*2, i**3) for i in xrange(1000000))
sa = np.fromiter(gen, dtype="i4,i8,f8")
tab = f.createTable(f.root, 'table', sa)
```

```
>>> tab
/table (Table(1000000,)) ''
  description := {
  "f0": Int32Col(shape=(), dflt=0, pos=0),
  "f1": Int64Col(shape=(), dflt=0, pos=1),
  "f2": Float64Col(shape=(), dflt=0.0, pos=2)}
  byteorder := 'little'
  chunkshape := (6553,)
```

Querying a Table

```
>>> tab[3]
(3, 6, 27.0)
>>> tab[3:2000]
array([(3, 6, 27.0), (4, 8, 64.0), (5, 10, 125.0), ...,
       (1997, 3994, 7964053973.0), (1998, 3996, 7976023992.0),
       (1999, 3998, 7988005999.0)],
      dtype=[('f0', '<i4'), ('f1', '<i8'), ('f2', '<f8')])

>>> tab[[3,100]]
array([(3, 6, 27.0), (100, 200, 1000000.0)],
      dtype=[('f0', '<i4'), ('f1', '<i8'), ('f2', '<f8')])

>>> [v[:] for v in tab.where("(f0 > 1) & (f2 < 100)")]
[(2, 4, 8.0), (3, 6, 27.0), (4, 8, 64.0)]
```

Modifying a Table

```
>>> tab[0] = (3, 3, 3.0)
>>> tab[:4]
array([(3, 3, 3.0), (1, 2, 1.0), (2, 4, 8.0), (3, 6, 27.0)],
      dtype=[('f0', '<i4'), ('f1', '<i8'), ('f2', '<f8')])

>>> tab[[1, 3]] = [(4, 4, 4.0)]*2
>>> tab[:4]
array([(3, 3, 3.0), (4, 4, 4.0), (2, 4, 8.0), (4, 4, 4.0)],
      dtype=[('f0', '<i4'), ('f1', '<i8'), ('f2', '<f8')])

>>> for row in tab.where("(f0 < 4) & (f2 <= 8.)"):
....:     row['f1'] = 0
....:     row.update()
....:
>>> tab[:4]
array([(3, 0, 3.0), (4, 4, 4.0), (2, 0, 8.0), (4, 4, 4.0)],
      dtype=[('f0', '<i4'), ('f1', '<i8'), ('f2', '<f8')])
```

Annotating you Datasets

```
>>> print tab
/table (Table(1000000,)) ''
>>> tab.attrs.TITLE = "sample data"
>>> print tab
/table (Table(1000000,)) 'sample data'
>>> tab.attrs.CLASS
'TABLE'
>>> tab.attrs.mycomment = "Enjoy data!"
>>> tab.attrs.complementary_data = np.array([3,2,3])
>>> tab.attrs.complementary_data
array([3, 2, 3])
```

Outline

Why compression?

- Files takes less space (the obvious reason).
- I/O speed can benefit a lot.
- If compression speed is good enough, it is a nice way to shelve arrays in-memory.

Two compression paradigms

Solid

- Data is compressed and decompressed as a whole.
- A compressed buffer must be decompressed completely before usage.
- The typical case is compressing a pickle.

Chunked

- Data is stored compressed in chunks and a chunk is decompressed only when it is needed.
- Typical case is HDF5 / NetCDF4 files (or compressed filesystems).

Outline

Solid compression

- Python comes with a series of codecs (compressor/decompressor) that are ready to use.
- One approach is to compress a pickle:

```
>>> a = np.linspace(0, 100, 1e7)
>>> pa = cPickle.dumps(a, cPickle.HIGHEST_PROTOCOL)
>>> a.size*a.itemsize, len(pa)
(80000000, 80000135)

>>> zpa = zlib.compress(pa, 9)
>>> len(pa), len(zpa), float(len(zpa)) / len(pa)
(80000135, 52946378, 0.6618286081642237)

>>> bpa = blosc.compress(pa, a.itemsize, 9)
>>> len(pa), len(bloscpa), float(len(bpa)) / len(pa)
(80000135, 7634771, 0.095434476454320099)
```

I/O with a compressed pickle

- Compressed pickles can be saved easily.
- Simply treat them as binary streams.

```
>>> with open("my_cpickle.bin", "wb") as f:
...:     f.write(zpa)
>>> with open("my_cpickle.bin", "rb") as f:
...:     zpar = f.read()
>>> assert zpa == zpar
```

Unpickling a compressed pickle

- Just decompress it first

```
>>> pad = zlib.decompress(zpar)
>>> assert pa == pad
>>> a2 = cPickle.loads(pad)
>>> np.alltrue(a == a2)
True
```

Sneak preview: Bloscpack

- An alternative is to use the `bloscpack` command line comprssion tool on the NPY/NPZ data

```
$ ls -sh test.npy
77M test.npy
$ time bloscpack compress --clevel 9 test.npy
bloscpack compress --clevel 9 test.npy  0.20s user 0.08s system 115% cpu
$ ls -sh test.npy.blp
668K test.npy.blp
```

- Compression ratio: 0.008470
- Not bad, but bear in mind that the example is a linear progression of numbers

Sneak preview: Bloscpack

- What about a more realistic example? (linear and periodic data)

```
$ ls -sh test.npz
153M test.npz
$ time bloscpack.py --clevel 9 test.npz
bloscpack compress --clevel 9 test.npz  1.17s user 0.31s system 139% cpu
$ ls -sh test.npz.blp
52M test.npz.blp
```

- Compression ratio: 0.337617
- Reminder `savez_compressed` resulted in
    - a filesize of `109M`
    - in roughly 22 seconds
- 47% smaller
- Much, much faster

Resource consumption for solid compression

Memory:

You need to book some spare memory to keep the compressed pickle.

CPU:

Compressors consume quite a lot of it, but you may always find a compressor that fits your needs.

For example, for a pickle of `np.linspace(0, 100, 1e7)`:

(all compressors with level 9)	memcpy	blosc	zlib	bzip2
final size (MB)	76	7.7	50	55
compress throughput (MB/s)	3500	3600	4.8	4.5
decompress throughput (MB/s)	3500	3500	120	9.9

Hardware: 2 x Intel E5520 @ 2.27GHz, 8 MB third level cache.

Outline

Chunked compression

- Data is stored compressed in chunks (on-disk or in-memory)
- Chunks are decompressed when needed only.
- HDF5 / NetCDF4 support this paradigm.
- Saves disk and memory resources and may, in some situations, even accelerate the I/O speed.

Examples with PyTables/HDF5

- PyTables includes support for a fair number of compressors: Zlib, Bzip2, LZO and Blosc.
- It also supports *shuffle*, an interesting filter designed to improved compression ratios.
- You can choose whatever combination that proves to be more convenient for your needs.

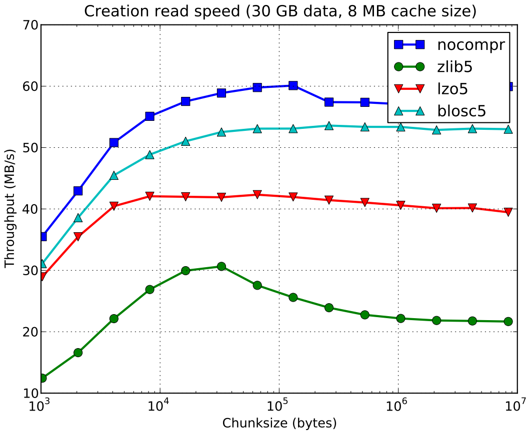
Querying compressed data

The dataset is a table with real data used in astronomy:

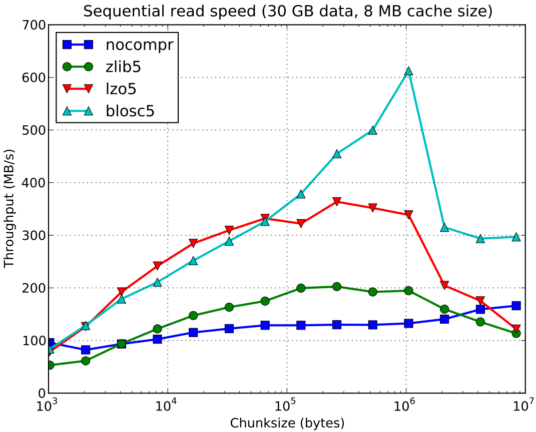
- 30 columns, most of them floating points and some ints
- Around 77000 entries
- Query: all entries where `ra` > 19 (3% selectivity)

(all compressors with level 5)	no compr	blosc	zlib	bzip2
table size (MB)	10	5.3	4.7	4.6
creation throughput (MB/s)	330	250	22	7.7
query throughput (MB/s)	170	140	38	10

Effect of chunked compression when writing large datasets



Effect of chunked compression when reading large datasets



When should you use compression?

- Your data has to be compressible (sparse matrices, time series, data with low entropy, ...).
- Whether your disk space is tight or your datasets are large.
- You want to optimize I/O speed.

Outline

Summary

- Pickle is the most basic, but still powerful, way to serialize Python data.
  - But it is mainly meant for small datasets and it is not portable.
- Relational databases are portable, mature and solid as a rock.
  - However, they do not interact well with NumPy and write performance is pretty lame.
- HDF5 shows best performance.
  - Python APIs interacts well with NumPy and are extremely portable.
  - They lack safety features.
- Using compression allows you to deal with more data using the same resources.
  - In general, they can save I/O time to disk.